Title	Towards a problem-driven approach to perspective-based reading
Author(s)	Chen, TY; Poon, PL; Tang, SF; Tse, TH; Yu, YT
Citation	The 7th IEEE International Symposium on High Assurance Systems Engineering Proceedings, Los Alamitos, CA., 23-25 October 2002, p. 221-229
Issued Date	2002
URL	http://hdl.handle.net/10722/48443
Rights	©2002 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Towards a Problem-Driven Approach to Perspective-Based Reading*

T. Y. Chen
School of Information Technology
Swinburne University of Technology
Hawthorn 3122, Australia
tychen@it.swin.edu.au

Pak-Lok Poon
Department of Accountancy
The Hong Kong Polytechnic University
Hung Hom, Kowloon, Hong Kong
acplpoon@inet.polyu.edu.hk

Sau-Fun Tang
Department of Finance
and Decision Sciences
Hong Kong Baptist University
Kowloon Tong, Hong Kong
ssftang@sinaman.com

T. H. Tse†
Department of Computer Science
and Information Systems
The University of Hong Kong
Pokfulam Road, Hong Kong
tse@csis.hku.hk

Y. T. Yu
Department of Computer Science
City University of Hong Kong
Kowloon Tong, Hong Kong
csytyu@cityu.edu.hk

Abstract

The quality of a requirements specification has a great impact on the quality of the software developed. Because of this, a requirements specification should be complete, correct, consistent, and unambiguous. Otherwise, defects may remain undetected, resulting in the delivery of a faulty software product to the users. Motivated by this, Basili et al. have developed the perspective-based reading (PBR) technique to help identify defects in requirements specifications.

In this paper, we propose a problem-driven approach for supporting the PBR technique. We also discuss the experience of applying our proposal to a real-life requirements specification.

Keywords: Classification-Tree Method, Defect-Based Reading, Perspective-Based Reading, Requirements Inspection, Software Inspection

*This work is supported in part by grants of the Research Grants Council of Hong Kong (Project Nos. CityU 1048/01E and HKU 7029/01E), a grant of the Innovation and Technology Fund (Project No. UIM/77), and a research and conference grant of the University of Hong Kong.

†**Contact author.**

1. Introduction

Requirements specifications, to be referred to as "specifications" for the rest of the paper, are undoubtedly important documents produced at the initial stage of the software development life cycle (SDLC). Since these documents form the basis for software design and implementation, they should be complete, correct, consistent, and unambiguous. Otherwise, any defect will be propagated to subsequent phases of the SDLC. At best, developers will eventually catch these defects, but at the expense of schedule delays and additional costs. At worst, the defects will remain undetected, resulting in the delivery of a faulty software product to users.

The cost-effectiveness of inspections in uncovering defects has been well discussed in the literature [3, 10, 11]. For example, Doolan [3] reports that industrial experience indicates a 30-time return on investment for every hour spent on inspecting specifications. Russell [10] reports a similar return of 33 hours of maintenance saved for every hour of inspection.

Consider some of the possible approaches or techniques used by *reviewers* to inspect a specification. The simplest is an ad hoc approach that has no formal or systematic procedure, but based largely on an individual's

Proceedings of the 7th IEEE International Symposium on High Assurance Systems Engineering (HASE'02)
1530-2059/02 $17.00 © 2002 **IEEE**

expertise and experience. A checklist approach represents an improvement, by providing reviewers with a list of items on which to focus. An alternative is defect-based reading, which helps reviewers focus on different classes of defects such as missing functionality and data type inconsistencies [8]. However, defect-based reading was originally developed for the specifications written in the software cost reduction notation, a formal notation developed by Heninger [6] for event-driven process control systems. This notation is not widely used in the commercial sector. In this regard, perspective-based reading (PBR) developed by Basili *et al*. [1, 12] has been considered a more widely applicable technique, as it can be used for inspecting specifications written in natural language. An important feature of PBR is that it allows reviewers in the inspection team to select their own perspectives (say, as a tester, developer, or user) when inspecting the specifications. Studies [1, 8, 12] report that inspection teams using PBR find more defects than teams using other approaches.

In this paper, we propose a "problem-driven" approach for supporting PBR. We recommend that, based on the reviewer's perspective *as well as* the characteristics of the problem domain of the specification, a specific method should be selected to establish a procedure for requirements inspection. The rationale is that, by selecting a method that suitably addresses the characteristics of the problem domain, we can increase the chances of detecting defects specific to these characteristics. We shall illustrate our approach by applying it to a real-life specification.

The rest of the paper is structured as follows. Section 2 outlines the original PBR technique developed by Basili *et al*. [1, 12]. Section 3 describes our proposal on the use of a problem-driven approach for supporting PBR. Section 4 discusses our experience of applying the proposal to a real-life specification. Finally, Section 5 concludes the paper.

2. Overview of perspective-based reading

Basili *et al*. [1, 12] argue that most inspection techniques do not help reviewers focus on particular aspects of a specification. Most techniques regard all the information in a specification as equally important because the document should define all the functional requirements and constraints. As a result, reviewers are left with an ill-defined responsibility of detecting all defects in the entire specification. This poses problems to reviewers and reduces their effectiveness in finding defects. Motivated by this observation, Basili *et al*. [1, 12] have developed PBR, a technique which operates under the premise that different information in a specification has different levels of importance for different uses of the document.

In general, PBR helps reviewers answer the following two important questions about the specifications they inspect: (*i*) What kind of information in the specifications should they check? (*ii*) How do they identify defects in that information? PBR focuses on the point of view of the "customers" of the specifications. For example, one reviewer may read from the point of view of the tester, another from the point of view of the developer, and yet another from the point of view of the user of the software. Each of these reviewers then produces a model that can be analyzed to answer questions based on the perspective. For example, Reviewer *A* in the team reading from the perspective of a tester would consider questions arising from activities related to *test suite* (the set of test cases used for software testing) design. Similarly, Reviewer *B* reading from the perspective of a developer would consider questions related to high-level system design, and Reviewer *C* representing the user would consider questions related to the completeness and correctness of the requirements with regard to system functionality. The assumption is that the union of their perspectives provides a comprehensive coverage of the specification, whereas each reviewer is responsible for a narrowly focused view of the specification, which should lead to more in-depth analysis of any potential defects in the specification. The validity of this assumption is confirmed by experiments involving the inspection of NASA documents [1].

In [12], Basili *et al*. have illustrated the application of PBR for inspecting a specification from the tester's perspective, by using equivalence partitioning (EP) and boundary value analysis (BVA). Basically, EP is a specification-based method that divides the input domain of a program into classes of data (each of which is called an *equivalence class*) from which test cases can be derived. In EP, input values that are treated the same way by the program are regarded as equivalent. To choose good test cases, it is better to select one from each equivalence class than to select all from one class [9]. On the other hand, BVA is a test case design method that often complements EP [9]. Rather than selecting an arbitrary element of an equivalence class, BVA assumes that a greater number of errors tend to occur at the boundaries of an input domain [9]. Because of this, elements at or near the boundaries should be selected to form test cases. Experiments such as those described in [1] confirm the viability of EP and BVA in detecting defects from the tester's perspective within the context of PBR.

3. A problem-driven approach to PBR

3.1. Rationale

We find the PBR technique innovative, particularly from the tester's perspective. One additional merit is that a test

suite can be generated from requirements inspection at an initial stage of the SDLC to be used for software testing at a later stage.

In general, any test suite construction method can be classified into either a *white-box* (*implementation-based*) approach or a *black-box* (*specification-based*) approach. The white-box approach aims to construct a test suite based on the information derived from the source code of the program. On the other hand, the black-box approach helps construct a test suite without the knowledge of the structure of the program. Obviously, white-box test suite construction methods cannot be used for PBR. This is because information on the program structure is not available when the specification is being inspected, since the program has not yet been developed.

Consequently, we turn our attention to the black-box approach. Basili *et al*. [12] have used two popular testing methods, equivalence partitioning (EP) complemented by boundary value analysis (BVA), to illustrate how to apply PBR to inspect specifications from the tester's perspective. We note that EP and BVA are but two of many specification-based methods for constructing test suites. The issue of whether other specification-based testing methods may serve a better purpose has not been fully addressed in the literature. In fact, a more general concern is the need to select specific methods in supporting PBR according to different characteristics of the problem domain.

Consider, for instance, a problem domain D that involves many input constraints.[1] Suppose the corresponding specification is RS_D. Since D has a lot of input constraints, chances are that RS_D will contain requirements defects related to such constraints. In this case, when applying PBR to inspect RS_D from a tester's perspective, requirements defects should be more easily uncovered if we select a test suite construction method (such as the classification-tree method [2, 5] and the category-partition method [7]) that takes explicit consideration of the input constraints. It is because the selected method specifically addresses the characteristics of the problem domain D that are potentially the source of requirements defects. In addition, consider a problem domain E whose corresponding specification is RS_E. If there are many different combinations of inputs in E, and each of these combinations results in a unique output, then cause-effect graphing [4] should be used for PBR from the tester's perspective. It is because cause-effect graphing makes explicit consideration of the interaction of inputs and the relationship between inputs and outputs. This property makes it an ideal candidate for PBR of RS_E.

[1] We illustrate this point with a program related to the application for credit cards by customers of a bank. Suppose the program has many inputs, in which "Employment Status" and "Monthly Salary" of a customer are two of them. In this case, a constraint between these two inputs is that, when "Employment Status = Unemployed", we must have "Monthly Salary = 0".

Because of the above, we propose a "problem-driven" approach for supporting the PBR technique. We shall discuss this in the context of a tester's perspective for the purpose of illustration. The approach should apply also to the perspective of a developer or user. Section 3.2 below describes our proposal in detail.

3.2. A sample procedure

Following on our argument in Section 3.1, we propose to use a ***PRO***blem-driven approach to ***PER***spective-based reading, abbreviated as PROPER. The acronym also reminds us that the selected method should be the most appropriate one for a given problem domain with respect to a given perspective. In this paper, we shall use the classification-tree method in the tester's perspective as an illustration.

Basically, PROPER consists of the following two distinct phases:

(1) Based on the characteristics of the problem domain P whose corresponding specification RS_P is to be inspected, select a specification-based test suite construction method M that specifically addresses these characteristics. For example, as explained in Section 3.1, the classification-tree method or the category-partition method can be selected for PBR if P contains a lot of input constraints. The rationale is that, by focusing on the explicit consideration of these characteristics of P, the selected method M will have a higher chance to detect any requirements defect in RS_P that is related to these characteristics.

(2) Apply the selected method M for PBR of RS_P, with a view to generating a test suite and to identifying any requirements defect in RS_P.

Note that, in phase (1) above, there are many possible specification-based test suite construction methods that can be selected. It is, however, not feasible to describe how to apply each of these methods to PBR in phase (2). For the purpose of illustration, we shall demonstrate the process using the classification-tree method (CTM) [2, 5]. CTM is chosen as the example because:

(*i*) The procedure $PROPER_{CTM}$ for applying CTM to PROPER can be modified without too much difficulty using other test suite construction methods.

(*ii*) Among the various test suite construction methods, CTM is relatively more complete, in the sense that it consists of well-defined steps that are straightforward to follow. Furthermore, it has been adopted by industrial projects such as that at the Daimler-Benz Group, a control system for the airfield lighting

of an international airport, and an integrated ship management system [5].

The procedure $PROPER_{CTM}$ provides questions tailored to each of its steps for creating a test suite using CTM. When creating a test suite, the reviewer answers a series of questions posed by $PROPER_{CTM}$. The underlying rationale is that, if the reviewer cannot answer the questions with respect to a particular requirement in the specification, then there is an anomaly associated with that requirement, although it may or may not be due to some defect. In this way, the reviewer can identify and fix the defects, if any, so that the requirements will better support the later phases of the SDLC.

The following is the procedure $PROPER_{CTM}$:

$PROPER_{CTM}$: A Procedure of PROPER supported by CTM

Decompose the specification into a number of units (known as "requirements units") that can be processed independently for the purpose of software testing. For each of these requirements units, follow the steps below to create a test suite. During the creation process, use the questions provided to identify any anomalies in the requirements unit.

Phase I: Preliminary Checking of Requirements Unit

Q.I.1. Does the requirements unit make sense from what you know about the problem domain?

Q.I.2. Does the requirements unit provide sufficient information for identifying the relevant and important aspects that may affect the resulting behavior of the system?[a]

Q.I.3. Based on the requirements stated in the specification and your knowledge of the problem domain, has any relevant and important aspect been omitted from the requirements unit?

Phase II: Creation of Test Suite

(1) Definition of classifications and classes. For every important and relevant aspect of the requirements unit, define it as a *classification*. Use this classification as a partitioning scheme to divide the input domain of the system into disjoint sets of elements called *classes*, such that all the elements in any class will behave similarly in the system.[b]

(2) Construction of a classification tree. From the information given in the requirements unit, determine the constraints among the classifications and classes defined in (1).[c] Based on these constraints, assemble the classifications and classes into a hierarchical structure, forming a classification tree.

(3) Construction of a test case table from the classification tree. Construct the corresponding test case table from the classification tree, so that a test suite can be created.

(4) Creation of a test suite. Identify all possible combinations of classes from the test case table. Each combination of classes represents a *test frame*. From every generated test frame, select one element from each of its classes to form a test case.

In steps (1) – (4) above, answer the following questions:

Q.II.1. Do you have sufficient information to define classifications and their associated classes?

Q.II.2. According to the information given in the requirements unit, have classes been defined so that no element in the input domain may appear in more than one class within the same classification?

Q.II.3. When assembling the classifications and classes to form a classification tree, does the requirements unit provide sufficient information for determining all the constraints among the defined classifications and classes?

Q.II.4. For every test frame, do you have sufficient information to select an element from each of its classes to form part of a test case?

Q.II.5. Are there other interpretations of the requirement that the developer may make on the basis of the description given? If so, will this affect the test cases you create?

Phase III: Preparation of Test Plan

For each test case generated, record the corresponding expected behavior of the system. In other words, how do you expect the system to respond to this test case you have just created? In this phase, the reviewer should ask the key question of whether the resulting behavior could be specified appropriately and without ambiguity.

[a] For example, in a typical credit-purchase application, "Customer Account Balance" is a relevant and important aspect that affects whether a credit-purchase transaction made by a customer should be approved.

[b] Refer, for instance, to the credit-purchase application in the footnote of Q.I.2 above. We define "Customer Account Balance" as a classification with two associated classes, "≤ 0" and "> 0". Normally, when "Customer Account Balance ≤ 0", the credit-purchase transaction will be rejected. On the other hand, when "Customer Account Balance > 0", the transaction may be approved, subject to other considerations such as the credit limit of the customer.

[c] Consider the credit-purchase application in the footnote of Q.I.2 again. Let "Credit Cardholder" and "Type of Credit Card" be two classifications for this application. Suppose the former has two associated classes "Yes" and "No", and the latter has two associated classes "Gold" and "Classic". A possible constraint is that, when "Credit Cardholder" takes the class "No", "Type of Credit Card" cannot take "Gold" or "Classic" as a class.

Readers may refer to [2, 5] for details about the use of CTM in generating a test suite. It should be noted that, in a particular application, $PROPER_{CTM}$ might be fine-tuned according to (*i*) the specific environment, and (*ii*) the

level of domain knowledge of the reviewer who applies it. Consider, for example, question Q.I.2 in Phase I of $PROPER_{CTM}$. Suppose the problem domain is the airline industry, and the reviewer knows that the type of aircraft is a relevant and important aspect affecting the system behavior. In this case, question Q.I.2 should be elaborated in more details by explicitly reminding the reviewer to check for the information on the aircraft type in the specification.

4. Experience in applying PROPER

In order to determine the viability of PROPER, we have done a case study on a real-life specification. Since the problem domain of this specification involves many input constraints, the reviewer has decided to apply CTM to support PROPER for the inspection of the document. This section outlines the way that PROPER has been applied, followed by a discussion on the results of the analysis.

4.1. Setting

The document to be inspected is a real-life specification prepared for an international company providing catering service for airlines, who prefers to remain anonymous and is referred to as ABC. The specification was produced for a meal ordering system (MOS) which helps ABC determine the types and the number of meals to be prepared and uploaded to each flight served by ABC. For the rest of the paper, the specification of MOS is referred to as RS_{MOS}. This specification contains various components such as narrative descriptions of the system, sample system screens, and reports. MOS has already been developed and released for production in ABC for several years.

We note that technical jargon unique to the airline industry is being used in RS_{MOS}, and hence it may not be easily understood by someone outside the project team to inspect the specification. Thus, an anomaly identified by a reviewer can of course be a real error in RS_{MOS}, but may also be due to a misunderstanding by the reviewer. We shall refer to all the anomalies as *potential defects*, and the real errors as *genuine defects*.

We have recruited two volunteers for our study. They are known as Participants *X* and *Y* in this paper. They have several years of experience working in the airline industry. Besides, Participant *X* has some practical experience using CTM, while Participant *Y* has been involved in the development and implementation of MOS. In the study, Participant *X* used PROPER to perform an inspection of RS_{MOS} from the tester's perspective and recorded all the potential defects. Participant *Y* then checked every potential defect to confirm whether it was genuine.

4.2. Observations and discussion

In our study, Participant *X* first decomposes RS_{MOS} into numerous requirements units that can be inspected or tested independently. For example, "Generation of Daily Meal Schedules" and "Maintenance of City Codes" are two of these decomposed requirements units. Among the requirements units, some are considered core requirements and others as auxiliary. For instance, "Generation of Daily Meal Schedules" is a core requirement, because it directly affects the type and the number of meals to be prepared and loaded onto each flight. On the other hand, the requirement "Maintenance of City Codes" is considered auxiliary, because it is only related to the maintenance of a reference table. We recommend that core requirements units should be reviewed before auxiliary units.

In this paper, we shall only focus on the core requirements unit related to the generation of daily meal schedules. This unit will be referred to as RU_{meal} for the rest of our discussion.

Phase I: Preliminary Checking of RU_{meal}

Based on his domain knowledge and experience, Participant *X* performs a preliminary check of RU_{meal}. The purpose is to determine whether RU_{meal} makes sense in general, and whether any relevant and important aspects that may affect the resulting behavior of MOS related to the generation of daily meal schedules have been omitted from RU_{meal}. In our study, no defect has been detected in the preliminary check.

Phase II: Creation of Test Suite

(1) Definition of classifications and classes

We use the following two examples to illustrate how classifications and classes are identified from RU_{meal}:

Example 1: Consider the following paragraph in RU_{meal}:

> ... Color of ''Flight Number'' displayed in the screen can be Red, Yellow, or Blue. These colors correspond to the outdated master flight schedules (MFSs), MFSs currently being used, and future MFSs, respectively. ...

In addition, it is stated elsewhere in RU_{meal} that only "current" MFSs will be used to generate daily meal schedules for a particular date. In this situation, Participant *X* defines [Effective Period of MFS] as a classification because of its effect on the generation of daily meal schedules. Furthermore, Participant *X* defines |Outdated|, |Current|, and |Future| as the associated classes of the classification [Effective Period of MFS]. ■

Example 2: Consider the following paragraph in RU_{meal}:

> ... Airlines can change the aircraft type, flight sector, or estimated time of departure (ETD) of a flight on a particular date even after the MFS has been announced. The system keeps an Exceptional Flight Schedule in addition to the MFS. This Exceptional Flight Schedule has higher priority over the MFS when creating Daily Meal Schedule. ...

From the above, Participant *X* defines [Existence of Exceptional Flight Schedule] as a classification because it is an important aspect to be considered when generating daily meal schedules. For this classification, Participant *X* defines |Yes| and |No| as the associated classes. ■

Following on in a similar way, Participant *X* defines a total of 15 classifications from RU_{meal}. Each of them has, on average, two or three associated classes.

When Participant *X* defines classifications and classes from RU_{meal}, he reports that a total of 13 potential specification defects have been detected. He then verifies with Participant *Y* and confirms that all these defects are genuine. In addition, Participant *X* observes that all these 13 defects can be grouped under two categories, depending on how they are detected from RU_{meal}. The first category includes those defects directly detected as a result of defining classifications and classes. Consider, for instance, the following paragraph in RU_{meal} related to the generation of daily meal schedules from MFSs:

> ... The only user-input for the function ''Generation of Daily Meal Schedules'' is the departure date. User can preview the Daily Meal Schedule of any given departure date. However, they can only create the Daily Meal Schedule following the last one created. ...

Participant *X* identifies the classification [Input Departure Date — Last Creation Date]. $|\leq 0|$ is a possible class, which will cause MOS to reject the departure date entered. When Participant *X* defines other classes for [Input Departure Date — Last Creation Date], he finds that RU_{meal} does not provide sufficient information for him to determine which of the following two alternatives should be chosen:

(*a*) define $|\geq 1|$ as the only extra class, or

(*b*) define two more extra classes, namely $|= 1|$ and $|\geq 2|$.

The problem is that RU_{meal} does not state clearly whether the departure date entered must be the day *immediately* following the last creation date. If it is stated clearly in RU_{meal} that this is the case, then Participant *X* will know that alternative (*b*) should be chosen. In (*b*), the definition of the class $|\geq 2|$ is useful in checking whether MOS is able to detect a wrong departure date (which does not immediately follow the last creation date) and reject it. Indeed, 6 of the 13 defects detected in this step are similar to this defect.

The second category of defects includes those *not* directly detected as a result of defining classifications and classes. Consider, for example, the following paragraph in RU_{meal}:

> ... Exceptional Crew Configuration has higher priority over Aircraft Configuration ... when creating Daily Meal Schedule. ... An Exceptional Crew Configuration record contains the Airline, Aircraft Type, Flight Number, Sector Number, ...

We know that information on sector numbers is kept in Exceptional Crew Configuration records. There is, however, no documentation in RU_{meal} (or indeed in the entire RS_{MOS}) describing what a sector number is, or how it relates to the generation of daily meal schedules. Because of this, it is obviously a specification defect.

Although this defect is detected when Participant *X* is defining classifications and classes from RU_{meal}, its detection is *not* directly due to the definition process itself. It is possible that a reviewer without the support of any systematic inspection technique can still detect this defect simply by reading RS_{MOS}. Similar defects account for 7 of the 13 defects detected in this step.

(2) Construction of a classification tree

In this step, Participant *X* identifies all the constraints among the classifications and classes defined in step (1) above, based on the information given in RU_{meal}. The identification of these constraints is required for the construction of a classification tree. We note that, in addition to the 13 defects detected in step (1), four additional defects are detected by Participant *X* in this step and they are confirmed to be genuine by Participant *Y*. Before we explain how additional defects can be detected via the tree construction, let us give an overview of the structure of a master flight schedule (MFS) and the structure of a classification tree.

Basically, an MFS contains numerous data elements for determining whether it should be selected for

generating a daily meal schedule. Consider, for example, the following two MFSs:

First MFS: `HK001 05/04/2002 − 05/06/2002`
`1.3.... 747 HKG → MEL ...`

Second MFS: `HK002 05/11/2002 − 05/24/2002`
`1234567 737 HKG → OSA ...`

The data elements, from top to bottom, and from left to right, correspond to the flight number, effective period, weekly flight pattern, aircraft type, and flight sector, respectively. Note that all date elements in MOS are in mm/dd/yyyy format.

In Example 1, it is mentioned that any MFS belongs to one of three types, namely outdated, current, or future. Given any MFS, its type is determined by comparing the date element "Effective Period" with the departure date entered online by the user. For instance, suppose the user enters a departure date "05/04/2002" into MOS. Then, the first MFS with flight number "HK001" is the current schedule, since the entered departure date falls within the effective period. On the other hand, the second MFS with flight number "HK002" is a future schedule, since the entered departure date is earlier than the effective period of that MFS.

Consider the weekly flight pattern "1.3...." of the first MFS, in which the numeric digits correspond to the days of the week. Such a pattern tells MOS that, within the effective period of the first MFS, this record should be selected for generating daily meal schedules on Monday and Wednesday only. This is because flight "HK001" will only depart on these two days of the week. Now, consider the weekly flight pattern "1234567" of the second MFS. It shows that "HK002" is a daily flight. It triggers MOS to select this record in generating daily meal schedules on every day of the week from "05/11/2002" to "05/24/2002".

From the above discussion, it is not difficult to see that test cases should be constructed for testing the behavior of MOS for outdated MFSs, current MFSs, and future MFSs with respect to the departure date entered. According to RU_{meal}, MOS will generate daily meal schedules for current MFSs but not outdated or future MFSs. In addition, for any *current* MFS, test cases should be constructed for testing each of the following situations:

(*a*) Its weekly flight pattern is "1234567", indicating that it is a daily flight. In this situation, no matter which day of the week the entered departure date is, this MFS will always be used to generate a daily meal schedule.

(*b*) Its weekly flight pattern is of any form other than "1234567", indicating that it is a non-daily flight. In this situation, the following two types of test case are generated:

(*i*) The first type corresponds to "normal" situations where at least one flight departs, by comparing the effective period with the weekly flight pattern of this MFS. An example is the first MFS above, with an effective period between Monday 05/04/2002 and Wednesday 05/06/2002 and having a weekly flight pattern "1.3....", which means Monday and Wednesday flights.

(*ii*) The second type corresponds to "abnormal" situations where no flight satisfying the weekly flight pattern can be identified from the effective period. An example is an MFS with an effective period between Monday 05/04/2002 and Tuesday 05/05/2002 but a weekly flight pattern ".....67", which corresponds to Saturday and Sunday. This type of test case is useful for testing whether MOS can recognize and handle abnormal situations.

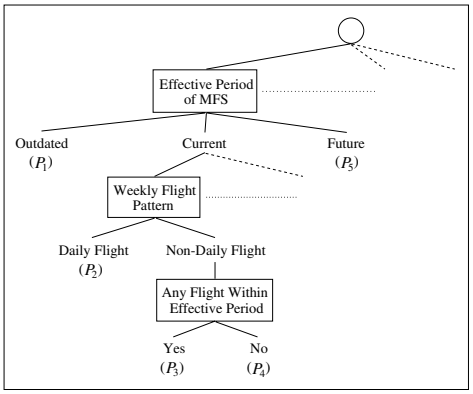

Figure 1. A partial classification tree ($\tau_{RU_{meal}}$) for RU_{meal}

Figure 1 shows a partial classification tree $\tau_{RU_{meal}}$ for RU_{meal}. In this figure,

- The circle at the top of $\tau_{RU_{meal}}$ represents the part of the input domain of MOS that is relevant to RU_{meal}.

- Classifications are enclosed in boxes whereas classes are not.
- The symbols enclosed in brackets represent paths of $\tau_{RU_{meal}}$ that contain some classifications and classes. For example, P_3 represents the path [Effective Period of MFS] — |Current| — [Weekly Flight Pattern] — |Non-Daily Flight| — [Any Flight within Effective Period] — |Yes|.

The basic idea of $\tau_{RU_{meal}}$ is to capture the constraints among the classifications and classes defined in RU_{meal}. For instance, given the predefined rules in constructing the test case table and the resulting test suite (see steps (3) and (4) in Phase II of $PROPER_{CTM}$), we are not allowed to construct a test case that contains the classes "Daily Flight" (in path P_2) and "Yes" (in path P_3).[2] In fact, paths P_2, P_3, and P_4 in Figure 1 correspond to the situations (*a*), (*b*)(*i*), and (*b*)(*ii*) above.

Having discussed the underlying data structures, we are now ready to explain how additional specification defects are detected via the construction of a classification tree. We do this through the following two examples:

Example 3: In fact, $\tau_{RU_{meal}}$ is part of the classification tree actually constructed by Participant *X* for RU_{meal} in our study. During the tree construction process, Participant *X* has to explicitly consider the constraints among the defined classifications and classes in step (2) in Phase II of $PROPER_{CTM}$. When considering the constraints between the classifications [Weekly Flight Pattern] and [Any Flight within Effective Period] and their associated classes with a view to constructing a classification tree, Participant *X* realizes that RU_{meal} does not specify how MOS should behave in response to situation (*b*)(*ii*) above (corresponding to the path P_4 in Figure 1). Note that, intuitively, this defect is difficult to be detected by simply defining classifications and classes in step (2) without considering their constraints. ■

Example 4: Consider the following description in RU_{meal} on Exceptional Crew Configuration records:

```
... Aircraft Configuration, such as the
number of crewmembers, is retrieved from
the Menu Planning System (MPS). However,
this information can be overridden
for flight schedules with additional
crewmembers.  The system keeps an
Exceptional Crew Configuration table
for this purpose.  Exceptional Crew
Configuration has higher priority over
Aircraft Configuration in MPS when
creating Daily Meal Schedule.  ... An
Exceptional Crew Configuration record
contains Airline, Flight Number, Number of
Crewmembers, ...
```

[2]The rules for constructing the test case table and the resulting test suite are fairly straightforward and hence will not be discussed in the paper. Readers may refer to [5] for details.

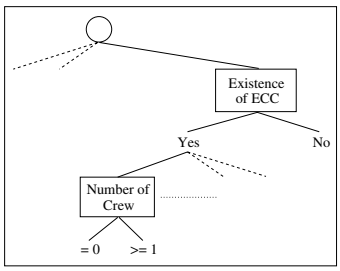

Figure 2. A partial classification tree for RU_{meal}

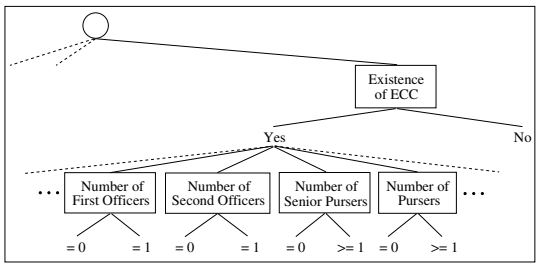

Figure 3. Another partial classification tree for RU_{meal}

The crew in a flight normally consists of the captain, no more than one first officer, no more than one second officer, the chief purser, several senior pursers, and several pursers. The actual composition of the crew will vary according to the aircraft type. With this domain knowledge, when Participant *X* reads the above description on Exceptional Crew Configuration (ECC) records, he does not know whether (*i*) only the total number of crewmembers on the flight is specified, or (*ii*) different numbers of crewmembers are specified for different types of crew. If case (*i*) is true, the partial classification tree in Figure 2 may be constructed. On the other hand, if case (*ii*) applies, then the partial classification tree in Figure 3 may be constructed instead. Again, this ambiguity defect is detected mainly due to the explicit consideration of constraints among classifications and classes during the construction of a classification tree for RU_{meal}. ■

We shall omit Phase III and steps (3) and (4) of Phase II in $PROPER_{CTM}$ because they are relatively straightforward and Participant X has not found additional defects in these steps/phases in our study.

5. Conclusion

There is no doubt that a specification must be complete, correct, consistent, and unambiguous, in order to reduce the chance of having defects in the resulting software product delivered to the end users. In this regard, requirements inspections remain a popular and effective technique in uncovering requirements defects.

In this paper, we have proposed an enhanced approach, known as PROPER, for requirements inspections. It is developed by supplementing the original PBR technique developed by Basili *et al*. [1, 12] with a problem-driven approach. The main contribution of our approach lies in stressing the need to select a specific method to support PBR by considering the characteristics of the problem domain of the specification to be inspected. The rationale is that, by selecting a method that suitably addresses the characteristics of the problem domain, we can increase the chances of detecting specification defects related to these characteristics. Our proposed approach has been validated in a case study of a real-life commercial specification. The results of the study show that PROPER is effective in detecting various kinds of defect.

We note that, although we have only illustrated the application of PROPER in the context of the tester's perspective in this paper, the problem-driven approach can also be applied to the perspective of the developer or user. In the developer's perspective, for example, if the problem domain of the specification involves a lot of decision and control flows, then decision tables, decision trees, or control flow diagrams may be used for PBR. On the other hand, if the problem domain involves complex data structures, then entity-relationship diagrams may be employed.

In conclusion, we find PROPER to be a technique worth further investigation. We are conducting a more comprehensive evaluation using a variety of real-life specifications.

Acknowledgement

We are grateful to the anonymous catering service company for providing the requirements specification in our study.

References

[1] V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgard, and M. V. Zelkowitz, "The Empirical Investigation of Perspective-Based Reading", *Empirical Software Engineering: An International Journal*, vol. 1, no. 2, 1996, pp. 133–164.

[2] T. Y. Chen, P. L. Poon, and T. H. Tse, "An Integrated Classification-Tree Methodology for Test Case Generation", *International Journal of Software Engineering and Knowledge Engineering*, vol. 10, no. 6, 2000, pp. 647–679.

[3] E. P. Doolan, "Experience with Fagan's Inspection Method", *Software: Practice and Experience*, vol. 22, no. 2, 1992, pp. 173–182.

[4] W. R. Elmendorf, "Functional Analysis Using Cause-Effect Graphs", in *Proceedings of SHARE XLIII*, New York, 1974, pp. 567–577.

[5] M. Grochtmann and K. Grimm, "Classification Trees for Partition Testing", *Software Testing, Verification and Reliability*, vol. 3, no. 2, 1993, pp. 63–82.

[6] K. L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application", *IEEE Transactions on Software Engineering*, vol. SE-6, no. 1, 1980, pp. 2–13.

[7] T. J. Ostrand and M. J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests", *Communications of the ACM*, vol. 31, no. 6, 1988, pp. 676–686.

[8] A. A. Porter, L. G. Votta, Jr., and V. R. Basili, "Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment", *IEEE Transactions on Software Engineering*, vol. 21, no. 6, 1995, pp. 563–575.

[9] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, New York, 2000.

[10] G. W. Russell, "Experience with Inspection in Ultralarge-Scale Development", *IEEE Software*, vol. 8, no. 1, 1991, pp. 25–31.

[11] G. M. Schneider, J. Martin, and W. T. Tsai, "An Experimental Study of Fault Detection in User Requirements Documents", *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 2, 1992, pp. 188–204.

[12] F. Shull, I. Rus, and V. Basili, "How Perspective-Based Reading Can Improve Requirements Inspections", *IEEE Computer*, vol. 33, no. 7, 2000, pp. 73–79.

[13] R. A. Weber, *Information Systems Control and Audit*, Prentice Hall, Upper Saddle River, NJ, 1999.